I'm not robot

Formcontrol vs ngmodel

Difference between ngmodel and formcontrolname. How to use ngmodel with formcontrol. Form control vs ngmodel. Difference between ngmodel and (ngmodel). Difference between ngmodel and formcontrol.

Angular Forms API: Understanding its Components and Use Cases ========== This post delves into the world of Angular Forms API, exploring how it enables the creation of complex forms. We'll cover Template-Driven Forms, the ngModel, ngForm, and ngSubmit directives, as well as Reactive Forms. **Template Driven Forms** Similar to AngularJs' ng-model directive, Template-Driven Forms provide two-way data binding between form inputs and view models. However, unlike AngularJs, this feature is now part of the main Angular framework. ### ngModel The ngModel directive enables form input value synchronization with view models. It's similar to AngularJs' ng-model but offers improved functionality and support for reactive forms. ### ngForm and ngSubmit Directives ngForm provides a way to group related form elements, while the ngSubmit directive handles form submissions. **Reactive Forms** Reactive Forms offer a different approach to form handling, using observables to manage form state. This approach is more flexible than Template-Driven Forms but also introduces new complexities. ### FormBuilder API The FormBuilder API provides a way to create reactive forms programmatically. ### Updating Form Values and Resetting Forms Forms can be updated by accessing the form's value map or by using the ngModel directive with a custom update function. Forms can be reset by calling the reset method on the form instance. **Advantages and Disadvantages of Both Form Types** Template-Driven Forms offer simplicity and ease of use, but they can become cumbersome for large forms. Reactive Forms provide more flexibility but require a better understanding of observables and form state management. ### Using Both Form Types Together Both form types can be used together to create hybrid forms that leverage the strengths of each approach. **Choosing Between Form Types** The choice between Template-Driven Forms and Reactive Forms depends on the specific requirements of your application. If you need a simple, straightforward form handling solution, Template-Driven Forms might be sufficient. However, if you require more flexibility and customization options, Reactive Forms are likely a better fit. ### Related Posts * [Angular Forms - what is it all about?](link) * [A large category of frontend applications are very form-intensive, especially in the case of enterprise development](link) Unless you add the `ngNoForm` attribute to your form element, you can start building an Angular Form. Let's create our first template-driven form! We'll see that there's a lot going on in this simple example. We've declared a basic form with two controls: first name and password. Both are mandatory fields, marked with the `required` attribute. We're using the `ngForm` directive to get a reference to the Angular Forms module, which tracks the overall value and validity state of our form. The `ngModel` directive is applied to each form control, binding to the corresponding HTML element and tracking its value and validity state. The `ngForm` parent directive interacts with its child `ngModel` directives, building a model of the whole form with all its field values and validity states. When we submit the form, the `onSubmitTemplateBased` method is called, only enabling the submit button if both required fields are filled in. Notice that our form submission doesn't trigger a backend HTTP POST request; instead, it calls the `onSubmitTemplateBased` method. We can access the native form submission event using `$event`. When a user interacts with a required field in Angular, the field turns red until they enter something. This is because Angular tracks three form field states: ng-touched or ng-untouched, ng-valid or ng-invalid, and ng-pristine or ng-dirty. These classes are applied to both the form and its controls, allowing for styling of error states at both individual control and whole form levels. The three state class pairs have specific meanings: * ng-untouched: The user hasn't interacted with the control (or form). * ng-touched: The user has interacted with the control at least once. * ng-valid or ng-invalid: The control's value is either valid or invalid. A form is considered touched if any of its controls are, and it's considered invalid if at least one control is. A form starts in a pristine state when initialized with backend data, which means no user modifications have been made. As the user makes changes, the form becomes dirty, and the ng-dirty class replaces ng-pristine. Touched and dirty states are related but distinct: touched refers to user interaction, while dirty indicates data modification. This separation allows for precise control over error state styling. In the example above, Angular tracks the whole form's validity to enable or disable the submit button. Much of this functionality is shared between template-driven and reactive forms. Using template-driven forms with Angular is a straightforward process that leverages its built-in functionality to handle form logic and validation. The key lies in understanding how Angular applies form directives to the view, creating an ngForm directive and registering validators automatically. Angular achieves this through implicitly defined form directives applied to the view, which include a form-level ngForm directive for each form in the template. This creates a form model that can be accessed using the myForm export, allowing for one-way or bi-directional data binding between the form controls and the component class. Template-driven forms are called as such because their validation and binding are setup in a declarative way at the level of the template, without requiring code in the component class. The ngModel directive plays a crucial role here, registering itself with the parent ngForm and validators if elements like required or maxlength are applied to the input. The ngModel directive also establishes bidirectional data binding by default, but its usage can be limited to one-way data-binding for specific scenarios. By using the one-way binding syntax, developers can initialize forms with pre-filled values without immediately reflecting changes in the component class. For cases where only form validation is required, without any type of binding, Angular provides an alternative approach that doesn't rely on ngModel's bi-directional data binding capabilities. Using ngModel without Binding: A Comparative Analysis If you want to utilize ngModel's validation and value tracking features without any kind of binding, you can do so using the following syntax: ngModel does not require any binding to function. Its primary purpose is to track a form control's value and validity state. However, this approach has its limitations. In template-driven forms, the form validation rules are typically stored in the HTML template. This makes it difficult to maintain and read as the template grows more complex. One alternative is to use the ReactiveForms module. Unlike template-driven forms, which require a separate module (FormsModule), reactive forms only require importing ReactiveFormsModule. Using the FormControl constructor, form control instances can be created individually. This results in a programmatic definition of the form model with all its controls and validation rules, generated at the component class level rather than within a template. The FormBuilder API provides an alternative notation for defining form models programmatically, which can be more concise for larger forms. Instead of directly calling FormGroup and FormControl constructors, the FormBuilder service is used to simplify the definition process using array notation. This approach makes it easier to define form models dynamically based on backend data, unit test business validation rules, and create custom validators by simply defining a function. Reactive Forms offer several advantages over template-driven forms. The component's template remains cleaner and focused on presentation logic, while business validation rules are moved to the component class for easier testing and modification. Dynamic form creation becomes more feasible based on backend data, and implementing custom validators is streamlined with the need to define only a function rather than creating a custom directive. Reactive Forms derive their name from providing an Observable-based API. Form controls and the entire form model can be viewed as continuous streams of values that can be subscribed to and updated in real-time. This allows for seamless integration with reactive programming concepts, making it easier to handle complex form interactions and updates within the application. Using RxJs operators, we can tap into the Form stream and apply functions like map and filter. For instance, we can convert first names to uppercase using map and filter out invalid values. This creates a new stream that we can subscribe to, providing a callback for UI updates. This approach makes it easy to implement advanced features like form drafts and undo/redo. We also have APIs for updating the whole form or individual fields. For example, we can create buttons to update the form value partially or fully. The FormGroup provides two methods: patchValue() for partial updates and setValue() for full updates. When using setValue(), we must provide values for all fields; otherwise, we'll get an error. To reset a form, we can use the FormGroup API, which resets everything back to pristine and untouched. We can mix Reactive and Template-Driven forms, as both are implemented with a FormGroup and FormControl instances under the hood. The main difference is that with Reactive Forms, we define the form model explicitly in our component class, whereas Template-Driven forms build the same form model behind the scenes using directives. Reactive Forms vs Template Driven Forms: Choosing the Best Approach When building forms in Angular, developers face a common dilemma: whether to use Reactive Forms or Template Driven Forms. While both approaches have their merits, Reactive Forms are generally considered the better choice for new applications. Reactive Forms offer several advantages over Template Driven Forms, including scalability and maintainability. They promote a clearer separation between business logic and presentation logic, making it easier to read and understand HTML templates. Additionally, Reactive Forms provide more advanced features, such as custom validation rules and support for Observable-based APIs. Template Driven Forms, on the other hand, are better suited for migrating existing AngularJs applications into Angular. They offer a familiar syntax that is easy to work with, but can lead to cluttered templates if not used carefully. Ultimately, Reactive Forms are more powerful and easier to use than Template Driven Forms. While it's possible to use both approaches together, it's generally recommended to stick with one or the other to avoid confusion. Choosing Between Angular Reactive and Template-Driven Forms ========== When building forms in an Angular application, developers have two primary options: reactive and template-driven forms. While both approaches have their strengths, adopting a single approach for the entire project can simplify form development and maintenance. For new projects or when working with a team, it's advisable to select one approach and stick to it throughout the application. In this case, we'll focus on Angular Reactive Forms as they offer more flexibility and power. **Learning Resources** If you're interested in learning more about Angular Reactive and Template-Driven Forms, check out our resources: * **Angular Forms In Depth**: A comprehensive course covering both reactive and template-driven forms. * **Angular for Beginners**: A beginner-friendly course that covers the basics of Angular and its form-related features. **The Power of Frameworks** Angular's framework nature provides a set of defaults and guard rails to follow when architecting new components. This ensures consistency and makes it easier to maintain complex applications. One such tool is the `@angular/forms` package, which offers powerful functionality like: * **Two-way data binding**: Seamlessly bind form elements to model properties. * **Disabling fields**: Easily hide or disable individual form fields based on user input. * **Data validation**: Implement basic validation rules for form fields. These features are particularly useful when dealing with complex form logic in your application. By leveraging the `@angular/forms` package, you can implement these functionalities in your own custom forms. **Example Component** Let's take a look at an example component that demonstrates the benefits of Angular Reactive Forms. This simple component allows users to type in data, displays a header label, and presents a fun message when "Unicorns" is typed in: ```typescript import { Component, Input } from "@angular/core"; @Component({ selector: "app-example-input", template: ` {{ placeholder }} You unlocked the secret unicorn rave!  {{ isSecretValue ? "You discovered the secret unicorn rave! They're all having a party now that you summoned them by typing their name" : "" }} `, styleUrls: ["./example-input.component.css"] }) export class ExampleInputComponent { @Input() placeholder = ""; value = ""; isSecretValue = false; } ``` By choosing a single approach and leveraging the power of Angular Reactive Forms, you can simplify form development, improve maintainability, and create more robust applications. You can leverage Angular's ControlValueAccessor interface to add form functionality to your input component, such as disabling the input or extracting data from it. This involves implementing four methods: writeValue, registerOnChange, registerOnTouched, and setDisabledState. To use these methods, you first need to provide them through a combination of providers array, NG_VALUE_ACCESSOR, and forwardRef in your component's setup. ```typescript import { forwardRef } from '@angular/core'; import { ControlValueAccessor, NG_VALUE_ACCESSOR } from '@angular/forms'; export const EXAMPLE_CONTROL_VALUE_ACCESSOR: any = { provide: NG_VALUE_ACCESSOR, useExisting: forwardRef(() => ExampleInputComponent), multi: true }; ``` You then pass this provider setup to your component's providers array: ```typescript @Component({ selector: 'app-example-input', templateUrl: './example-input.component.html', styleUrls: ['./example-input.component.css'], providers: [EXAMPLE_CONTROL_VALUE_ACCESSOR] }) export class ExampleInputComponent implements ControlValueAccessor { ``` This will allow you to use the writeValue, registerOnChange, registerOnTouched, and setDisabledState methods in your component. To enhance your property management, consider utilizing a setter, getter, and private internal value. Initialize the internal value as `any = null;` and implement getter and setter methods for `value()`. The getter method should return the current `_value`, while the setter method updates the `_value` only when the new value differs from the existing one. To facilitate seamless rendering of your component, you may need to provide a `ChangeDetectorRef` in the constructor and manually check for updates in the `writeValue()` method. This ensures that change detection occurs correctly even when updating values externally. For implementing disabled states, follow a similar approach by adding setter, getter, and setDisabledState() methods. Mark the component for checking whenever the disabled state changes to ensure seamless rendering. It's worth noting that setting disabled state is optional for ControlValueAccessor implementation. You can still use ngModel and formControl without this method implemented. To implement `registerOnChange()`, store the passed function in your class instance and call it whenever the user updates data. This allows you to update ngModel accordingly. Storing and Calling Functions. The code snippet demonstrates how to store a function but doesn't cover calling it after storage. To use the stored function, you'll need to call it with updated values on each update. For instance, if you're expecting an input value to change, you'd want to add it to the output of the input. You can do this by using the (change) event handler. Registering OnTouched. Just like storing a function and calling it to register changes, you'll need to register when a component has been "touched" or not. This informs your consumers when a component has had user interaction or not. To achieve this, use the onTouched method, which should be called whenever the user interacts with your component. Consuming the Component. Once you've implemented these features, you can start using them in your application. First, import FormModule and ReactiveFormModule into your AppModule for ngModel and formControl support respectively. Then, add a formControl item to your parent component and pass it to ngModel or formControl. Inspecting the Value. Finally, you can inspect the value of the input directly from the parent component by using formControl or ngModel. This allows you to see the current value of the input in real-time. Angular CSS masters use classes like ng-pristine and ng-dirty to reflect state changes on inputs. These classes don't appear with [(ngModel)], but do show up with [formControl] or [formControlName], thanks to registerOnChange and registerOnTouched functions. To check form control errors directly from a component, we need to assign the provider manually in the constructor. This involves making changes to the previous form input implementation, setting the value accessor directly, and utilizing @Self and @Optional decorators for dependency injection. With ngControl in hand, you can access the formControl using ngControl.control. In ngOnInit(), the FormControl should be available, allowing you to check when errors are applied and more. This provides a ton of different props to access for the control's metadata. get errors() method can be used with ngControl object to get touched and errors from control. This method is then referenced in template using [class.redtext]="errors" directive which changes class based on errors value. Component implementation uses FormControl from Angular forms library, importing necessary modules and validators. It's possible to add custom validators for FormControl. Validators are available through the Angular Forms Library. FormControl with Validators required can be used to validate form inputs, providing a wide range of built-in validators and allowing custom validators to be made. This makes it easier to ensure consistency across form components by using feature-rich APIs like formControl and ngModel. The article aims to provide in-depth insight into using these features, enabling developers to create more consistent form components that feel familiar to users.